Test Methods for Score-Based Interactive Music Systems

Clément Poncelet Sanchez, Florent Jacquemard

Test Methods for Score-Based Interactive Music Systems

Clément Poncelet
DGA & INRIA
UMR STMS (Ircam/CNRS/UPMC), Paris, France.
clement.poncelet@ircam.fr

Florent Jacquemard
INRIA
UMR STMS (IRCAM/CNRS/UPMC), Paris, France.
florent.jacquemard@inria.fr

ABSTRACT

Score-Based Interactive Music Systems (SBIMS) are involved in live performances with human musicians, reacting in realtime to audio signals and asynchronous incoming events according to a pre-specified timed scenario called a mixed score. This implies strong requirements of reliability and robustness to unforeseen errors in input.

In this paper, we present the application of formal methods for black-box conformance testing of embedded systems to SBIMS's. We describe how we have handled the 3 main problems in automatic testing reactive and realtime software like SBIMS: (i) the generation of relevant input data for testing, including delay values, with the sake of exhaustiveness, (ii) the computation of the corresponding expected output, according to a given mixed score, (iii) the test execution on input and verdict.

Our approach is based on formal models compiled from mixed scores. Using a symbolic checker, such a model is used both for (i), by a systematic exploration of the state space of the model, and for (ii) by simulation on a given test input. Moreover, we have implemented several scenarios for (iii), corresponding to different boundaries for the implementation under test (black box). The results obtained from this formal test method have permitted to identify bugs in the SBIMS Antescofo.

1. INTRODUCTION

Interactive music systems (IMS) presented in [1] are involved in live music performances with human musicians. They work by coupling functionalities of artificial listening, in particular score following and tempo detection, and of reactive systems, for synchronizing their outputs to musician inputs. In the case of SBIMS, all these activities are performed in realtime following a pre-specified timed scenario called a mixed score, written in a *domain specific language* (DSL).

During an instrumental performance, when a musician does a mistake, the piece must and will continue. However, IMS practitioners know that a crash or misbehavior of an IMS can jeopardize a mixed instrumental-electronic performance. In order to avoid unforeseen errors of an IMS at runtime, and meet listeners' expectations, it is important to be able to explore, statically, its reactions to possible musician's interpretations, and check that they conform to the behavior specified in the given mixed score. This difficult task is complicated by high unpredictability of musicians' inputs and hard temporal constraints (due in particular to the strong requirements of audio computing platforms).

A traditional and manual approach is to rehearse with musicians. However time is precious during a rehearsal, and its purpose is usually more to solve musical questions than to fix bugs. It is also possible to listen to an IMS playing with some recordings of musicians, checking that the system is not crashing and that the result *sounds* in a satisfiable way. The problem with this approach is that, on the one hand, the test input is not complete (it just represents one or a few particular performances), and on the other hand the verification of the outcome is not rigorous.

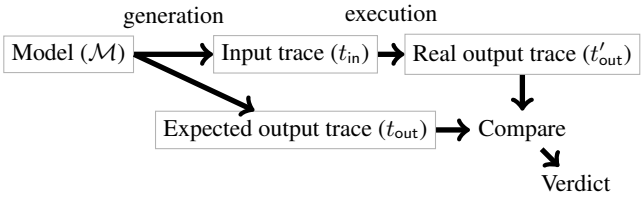

Figure 1: *Principles of Model Based Testing*

Several formal methods have been developed for automatic conformance testing of critical embedded software, see e.g. [2]. The principle (Figure 1) is to execute a real *implementation under test* (IUT) in a testing framework. When the source code of the IUT is not known and only its input and output are observed, we call it *black-box* testing. In conformance *model-based testing* (MBT), a formal specification, or model $\mathcal{M}$ of the system is written (in general manually) and used to generate automatically some test data. This comprises input test data t_{in}, sent to the IUT, and theoretically expected output test data t_{out}, computed from t_{in} using $\mathcal{M}$. The latter t_{out} is then compared to the real output test data t'_{out}, obtained from the IUT when it receives t_{in}, in order to produce a test verdict. This procedure is iterated on a large base of pairs $\langle t_{\mathsf{in}}, t_{\mathsf{out}} \rangle$, which is generated, for exhaustiveness purposes, according to a user specified *covering criteria*, expressed as a formula referring to elements of $\mathcal{M}$. This provides a rigorous mathematical framework, increasing the confidence in the tested systems and reducing test costs. For realtime

THIS WORK HAS BEEN PARTLY SUPPORTED BY THE ANR PROJECT INEDIT (ANR-12-CORD-009)

systems such as communication protocols, transportation control *etc*, like for IMS, time is a semantical issue (not just a measure of performance), and it must be included in test data, which become timed traces.

This paper presents a study of the application of MBT methods to the SBIMS Antescofo, developed at Ircam and used regularly in concerts. This system shares several characteristics of the reactive and realtime systems usually targeted by MBT, but also has its own specificities. In particular, the formal specification of the IO behavior of the system is produced automatically from a given mixed score, using a front-end compiler into an ad hoc *intermediate representation* (IR). This is in contrast with usual MBT case studies where the specification has to be written manually by an expert in formal methods. After a brief presentation of Antescofo and its score DSL (Sections 2.1 and 2.2), the test data is presented in Section 2.3. Then we describe some test scenarios that we have used for Antescofo (Section 2.4). Some are running in real time, some running in a fast forward mode thanks to implementation of virtual clocks in Antescofo. The automatic construction of formal model, using IR, is presented in Section 3. In particular, we show how we used the symbolic model checker Uppaal [3] based on *timed automata* model, and its extension CoVer [4] for the production of test input data with covering criteria.

The test outcome permits us to prepare concerts, by simulation of covering sets of fake musician performances (the test input t_{in}) derived from a given mixed score. Moreover, it increases the guarantee on the reliability on the system, with a systematic analysis of test outcome, exploited for debugging.

2. TEST FRAMEWORK FOR A SBIMS

We present in this part the principles of a testing framework that we have developed for the SBIMS Antescofo [1], following the approach depicted in Figure 1.

2.1 Antescofo

Collective music performance involves several complex and sometimes implicit activities. The system Antescofo aims at acting as an electronic musician interacting with human musicians, implementing these behaviours. For this purpose, the system takes as input **a mixed score** which describes in the same file some musician and electronic parts. During a performance, the system synchronizes the electronic parts to the musician's ones: it aligns in realtime the performance of human musicians to the score, handling possible errors, detects the current tempo, and plays the electronic part, following the detected tempo. Playing is done by passing by messages to an external audio environment such as MAX or PureData. A popular particular case of this behavior is automatic accompaniment [5].

Antescofo is therefore a *reactive* embedded system, interacting with the outside environment (the musicians), under strong timing constraints; the output messages must indeed be emitted at *the right moment*, not too late but also not too early. Figure 2 shows the Antescofo architecture. Our electronic musician is composed of two units. A *listening machine* (LM) receives an audio or midi stream from a musician and detects in realtime his position in the mixed score. This score following feature is coupled with a function of tempo inference based on Large's algorithm [6]. The positions and instantaneous tempo detected by the LM are sent to a *reactive engine* (RE) which schedules the electronic actions to be played and emits *on time* messages to the audio environment. Note that the information exchange between LM and RE is discrete, as well as the output of the system (messages sent).

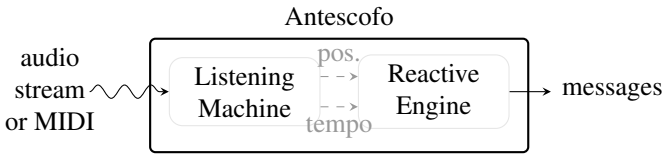

Figure 2: *Architecture of Antescofo*

2.2 Mixed Scores DSL

The mixed scores of Antescofo are written in a textual reactive synchronous language describing the electronic accompaniment as reaction to the detected instrumental events. A simplified extract of the score of *Einspielung I* [2] by Emmanuel Nunes is presented in Figure 3. This piece for violin and electronics will be used as a running example in this paper.

We give here an *abstract syntax* corresponding to a small part of this language, following our needs for presenting our test framework. The reader can find more complete descriptions in [7, 8].

Formally, an Antescofo *mixed score* is a finite sequence of input events $e_1, \ldots$, each event being bound to a finite

[1] `http://repmus.ircam.fr/antescofo`

[2] `http://brahms.ircam.fr/works/work/32409/`

```
bpm 144
note  D4 1/7  event1
      0  a0
      group 0  g1  @loose @global
      {
       0       a1
       1/7     a2
       1/7     a3
       1/7     a4
       1/7     a5
       1/7     a6
       1/7     a7
      }
chord  (  B3b  D4 )  1/7  event2
chord  (  E4   D4 )  1/7  event3
chord  (  D5#  D4 )  1/7  event4
chord  (  A4   D4 )  1/7  event5
chord  (  C4b  D4 )  1/7  event6
chord  (  G4   D4 )  1/7  event7
```

Figure 3: *Simplified extract of Einspielung in Antescofo*

sequence of triggered actions called $act(e_i)$. In the following, a finite sequence of actions is called *group*, and $act(e_i)$ is called *top-level* group triggered by e_i.

An *event* e_i is a tuple $\langle i, c, d, g\rangle$ made of the unique identifier i (event number), some event data c, the event's duration d (also denoted $dur(e_i)$), expressed either in number of beats of tempo or milliseconds (ms), and group g triggered by e_i. The event data contains information such as the event kind (note, chord, trill...), and pitch values. An important point here is that on detection of an event, the LM will return the id i to the RE (and not simply the pitch). Note that all values in Antescofo, including durations, are expressions which can possibly contain variables (global or local to groups) and functions.

An *action* is a pair $a = \langle d, g\rangle$ where d is a delay (in beats or ms) and g can be either an atom or a group. An *atom* is a control message sent to an external audio system – MAX or PureData or a computation of the form $x := exp$ where x is a variable and exp an expression. Note that with the above recursive definition, the groups can be nested arbitrarily, in order to reflect some musical intention. Moreover, every action is contained in a group, called its *container*.

The delay d in $a = \langle d, g\rangle$ is the time to wait before starting to play g, after the trigger of a has been detected (if it is an event) or started (if it is an action). The *trigger* of an action a with container g' is defined as follows. If a is not the first action of g', then its trigger is the action preceding a in g'. Otherwise, either g' is a top-level group $act(e_i)$ and the trigger of a is e_i, or g' is in an action $a' = \langle d', g'\rangle$ called *parent* of a and the trigger of a is the trigger of a'.

Some high-level attributes can be added to groups to express an expected behavior for musician-electronic synchronization and error handling, corresponding to a particular musical situations [5]. In this paper, we shall consider, for illustration purposes, a small sub-set of attributes : two synchronisation attributes:

loose: Synchronization on tempo. Only the tempo is used to compute the delays of the group's actions.

tight: Synchronization on events. Every action in the group is bound to the closest event.

and two error management attributes. In Antescofo, an error is a missing event (note), either because the musician did not play it or else because the LM did not detect it (e.g. because it is not well tuned).

local: Skip. If the triggering event is missing, the actions of the group are skipped.

global: Immediately. The actions are started immediately at the detection that the triggering event is missing.

Roughly, the synchronisation attribute expresses how smoothly (for loose) or not (for tight) the electronic part should be played. The error management specifies the importance of the actions. Figure 4 illustrates Antescofo's behavior for various compositions of attributes for the group in the extract of mixed score of Figure 3. Note that in the score, the attributes loose and global have been chosen.

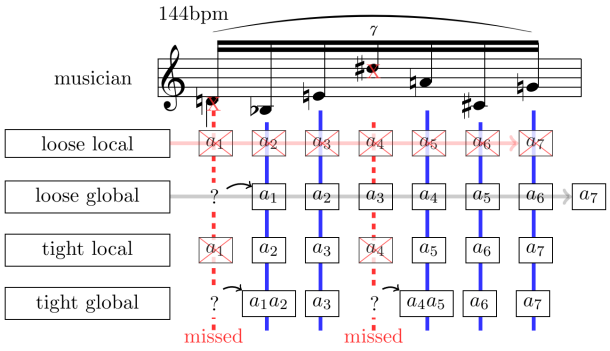

Figure 4: *Attribute specification (e1 and e4 missed)*

2.3 Test Input and Output Data

What is the form of the input data send to the SBIMS for testing its behavior, as well as the output data collected for analyzing the results of tests?

Basically, a test input is a trace of events representing a musician performance and an output is a trace of actions representing the electronic accompaniment generated in reaction to the input. For reactive and realtime IMS, time is a semantics property, and the dates at which events and actions are played must be included in test input and output data (timestamps). This is in contrast with functional programs, computing output from the given input, for which it is sufficient to consider, for testing, untimed data values.

A grand challenge for the design and implementation realtime embedded systems (including IMS) is to reconciliate two time units [9]: the time of the environment and the time of the platform. The first is the *physical time*, expressed in milli-seconds. The second is a logical time used by the system in its computations. For IMS, the logical time unit is the number of beats, it is called *relative time* in the rest of the paper. Hence, in IMS, reconciliation of the times of environment and platform is done through tempo.

A *test input trace* t_{in} is a finite sequence of triples $\langle i, d, T\rangle$ made of an event identifier i (pointing to an event e_i in a given mixed score), a duration value d expressed in relative time (like in the score), and the instant tempo T between e_i and the next event in the trace, expressed in beat per minute. Note that *missing events* can be specified, by absence, in t_{in}: the event e_i is missing in a t_{in} of the form $\ldots, \langle i-1, d_{i-1}, t_{i-1}\rangle, \langle i+1, d_{i+1}, t_{i+1}\rangle, \ldots$

As an example, from a mixed score $e_1, e_2, \ldots$ we can directly generated the so called *ideal trace* $\langle 1, dur(e_1), T\rangle$, ..., where T is the tempo specified in the score [3] (see Figure 5). This trace corresponds to the performance of a robot, playing exactly the notes and durations specified. An *expected output trace* t_{out} (resp. *real output trace* t'_{out})

$\langle 1, 1/7, 144\rangle \cdot \langle 2, 1/7, 144\rangle \cdot \langle 3, 1/7, 144\rangle \cdot \ldots \cdot \langle 7, 1/7, 144\rangle$
$\langle a_0, 0\rangle \cdot \langle a_1, 0\rangle \cdot \langle a_2, 1/7\rangle \cdot \langle a_3, 2/7\rangle \cdot \ldots \cdot \langle a_7, 6/7\rangle$

Figure 5: *Ideal input and expected output for Fig. 3*

is a finite sequence of pairs $\langle a, d\rangle$ made of an *atom* (as de-

[3] If the tempo changes in the score, then it is changed accordingly in the ideal trace.

fined in Section 2.2) and its date d expressed in relative time (resp. in physical time).

A *test case* (see Figure 1) is a pair made of an input trace t_{in} and the corresponding expected output trace t_{out}.

Related models of performances

Time-warps [10], *Time-Maps* (Jaffe 1985), *Time-deformations* (Anderson and Kuivila 1990), are continuous and monotonically increasing functions used to define either variations of *tempo* or variations of the duration of individual notes (*time-shift*). Some models of performance [10, 11] are defined by combination of these two transformations, defined independently. Our input test trace format is a discrete version of such models, where the tempo variations and time-shifts are defined respectively in the third and second component of entries $\langle i, d, T\rangle$. An important difference with [10,11] is the possibility to have missed notes in input traces.

Input trace fuzzing and generation

Thanks to these models, generating input traces scripting musical performances is not difficult. One can start with the ideal trace, generate arbitrary tempo values (e.g. defined by a tempo curve) and add some fuzz to events durations (time shifts) and missing events. The obtained traces are well suited for testing in the preparation of concerts. Another method for generating more exhaustive sets of input traces, suitable for debugging, is presented in Section 3.4.

Generating the corresponding expected output traces t_{out} is a more difficult problem: it wouldn't make sense to use the system under test for this purpose, and we need instead a formal reference of the timed behavior expected for the system. As explained in introduction, we follow a model-based approach (MBT) to tackle this problem, where a model $\mathcal{M}$ is used to compute t_{out} from t_{in} ($\mathcal{M}$ is also used to generate t_{in}); this is detailed in Section 3.

2.4 Test Execution

How can we execute given test cases on the SBIMS?

The execution of a test case $\langle t_{\mathsf{in}}, t_{\mathsf{out}}\rangle$ is somehow a monitored simulation of a performance. It consists in sending the events in the input trace t_{in} to the SBIMS, with their durations, and collect a *real output trace* t'_{out} by monitoring and time-stamping (in physical time) all output emissions of the SBIMS during the execution. The latter is then automatically compared to t_{out} to produce a test verdict.

The problem is more tricky that it seems due to the data flow in Antescofo, its modular nature (Figure 2) and the relative time unit used in test cases. We present below several scenarios for testing different parts of the system.

2.4.1 Testing the RE

This scenario is performed on a standalone version of Antescofo equipped with an internal *test adapter* module. The adapter iteratively reads one element $\langle i, d, T\rangle$ of a file containing t_{in}, converts d into a physical time value $d' = \frac{d.6.10^4}{T}$ (remember that delays are expressed in relative time in t_{in}), and waits d' ms before sending i and T to the RE. More precisely, it does not physically wait, but instead notifies a *virtual clock* in the RE that the time has flown of d' ms. This way the test does not need to be executed in realtime but can be done in fast-fordward mode. This is very important for batch execution of huge lists of test cases.

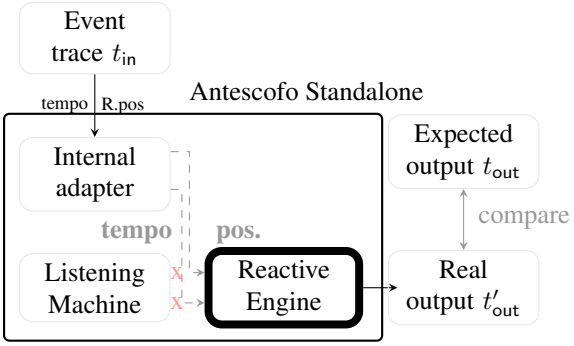

Figure 6: *Testing scenario of Section 2.4.1.*

The messages sent by RE are traced in t'_{out}, with timestamps in physical time (this functionality is built in the current RE). Finally, the timestamps in t_{out} are converted from relative time to physical time using the tempo values in t_{in}, in order to be comparable to t'_{out}.

In this scenario, the IUT is the RE (the LM is idle).

2.4.2 Testing the RE with tempo detection

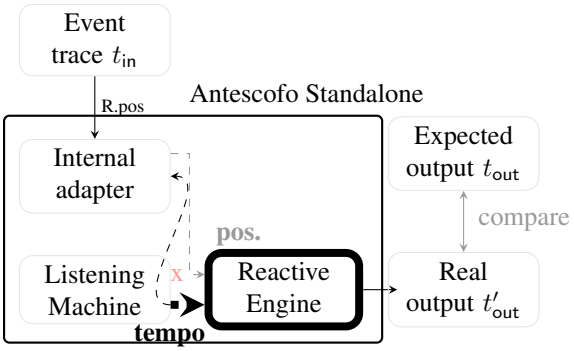

Figure 7: *Testing scenario of Section 2.4.2.*

In this second scenario the tempo values T are not read in t_{in} by the adapter but instead inferred by the LM (the adapter is calling an appropriate method in LM). The rest of the scenario is similar to Section 2.4.1. The values of detected tempo are stored by the adapter and used later to convert the dates in the expected trace t_{out} from relative to physical time. In this case, the IUT is somehow the RE plus the part of the LM in charge of tempo inference.

2.4.3 Testing the whole SBIMS as a Blackbox

This scenario is the most general. It is executed in a version of Antescofo embedded into MAX (as a MAX patch), using an adapter which is another MAX patch. The adapter iteratively reads triples $\langle i, d, T\rangle$ in a file containing t_{in}, and plays them as MIDI events, using the duration d converted to physical time with T. The audio stream generated is then sent to the LM, and the output of the RE is traced in t'_{out} as before.

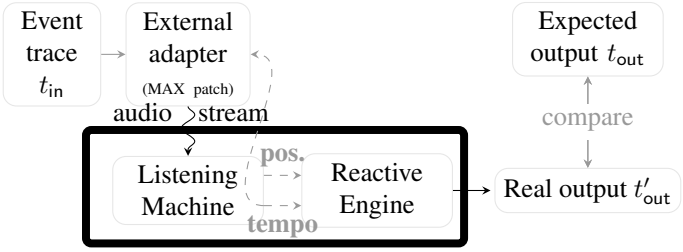

Figure 8: *Testing scenarios of Section 2.4.3.*

Note that here, the RE uses the tempo values detected by the LM, which will differ from the tempo values in t_{in} [6]. Therefore, the former are saved by the adapter (in MAX the detected tempo is available as an outlet of the antescofo~ patch), and used later to convert the dates in t_{out} from relative to physical time like in Section 2.4.2. In this realistic scenario, the IUT is the whole SBIMS.

In an alternative scenario, the adapter ignores the tempo values in t_{in} and uses tempo detected by the LM, similarly to Section 2.4.2. Note that in these scenarios, the tests are executed in realtime and not in a fast-forward mode like in Sections 2.4.1, 2.4.1.

2.5 Test Verdict

How can we check that the real output trace t'_{out} is correct? When t_{out} is not known, we are left to listen the execution of the IMS on t_{in}, in extenso, for instance using the framework presented in Section 2.4.3, and decide subjectively whether we are satisfied with it. This manual solution is not rigorous and also tedious when one need to consider many different t_{in} for covering purposes.

When t_{out} is known, we just need to compare pointwise the conversion of t_{out} to physical dates to t'_{out}, with a fixed error bound δ to dealing with latency, and report differences. This raises the following important question: *How can we systematically compute the expected t_{out} from t_{in}?* This is going to be answered in the next part of the paper.

3. MODEL BASED CONFORMANCE TESTING

We report here the use of state-of-the art MBT models, techniques and tools for testing the SBIMS Antescofo in the framework presented in previous Section 2.

3.1 Generalities on Model Based Testing

Figure 9 depicts in its higher half a reactive system's IUT interacting with an environment RealENV, and in its lower half, two formal specifications of the latter, resp. $\mathcal{S}$ and $\mathcal{E}$.

The *behavioral specification* $\mathcal{S}$ of the system is a formal description of its reactions to the outside environment. In our case, it is the function producing t_{out} given t_{in}.

The *environment model* $\mathcal{E}$ is a formal description of what can be expected from the environment. In our case, it is the definition of the set of all possible t_{in}, *i.e.* all the potential interpretations of musicians to be tested.

Note that since IMS are realtime systems, we need to express time in $\mathcal{E}$ and $\mathcal{S}$, like in: *"one message m has to be emitted one beat after the first event e_1 of the musician"*.

The *conformance* of the IUT to the specification $\mathcal{S}$ *wrt* $\mathcal{E}$ is defined as the inclusion of the set of real output traces t'_{out}, obtained by the execution of all $t_{\text{in}} \in \mathcal{E}$ against the IUT, into the set of expected output traces $t_{\text{out}} = \mathcal{S}(t_{\text{in}})$ with $t_{\text{in}} \in \mathcal{E}$. As time values are included in the traces, conformance ensures the *time safety* of the IUT on the test cases.

3.2 Antescofo Intermediate Representation

How can we write formal specifications $\mathcal{E}$ and $\mathcal{S}$ for testing the SBIMS Antescofo on a given mixed score s? Actually, this is the exact purpose of the score! Therefore, in our test framework we generate automatically from a score two formal specifications $\mathcal{E}$ and $\mathcal{S}$ of the expected behavior of the musicians and Antescofo, exploitable by testing tools. This prevents us opportunistically from the burden of an initial phase of manual specification by experts, generally needed for the testing and verification of embedded systems. Hence the automatic production of $\mathcal{E}$ and $\mathcal{S}$ is a convenient feature, typical of IMS testing.

We use a front-end compiler transforming an Antescofo mixed score s into a *medium level* executable intermediate representation $IR(s)$. The model $IR(s)$ will be furthermore translated into the timed automata formalism in order to use tools dealing with such models (Sections 3.3, 3.4). This approach is similar to the use of Ecode for the Giotto language [12] in order to ensure portability and predictability (determinism), both in timings and functionality. We present here a simplified graphical version of the IR designed for Antescofo [13].

An IR is a finite set (called *network*) of finite state machines extended with variables and durations (EFSM), communicating synchronously with some symbols taken from a finite alphabet Σ. Some example of EFSMs can be found in Figures 10 and 13. We let $\Sigma = \Sigma_{\text{in}} \uplus \Sigma_{\text{out}} \uplus \Sigma_{\text{sig}}$ where Σ_{in} and Σ_{out} are respectively the sets of the event ids and atomic messages of s (as described in Section 2.2), and Σ_{sig} contains internal signals presented below. Every transitions of the EFSMs is labelled by one of $\sigma!$ (emission of a symbol $\sigma \in \Sigma$), $\sigma?$ (reception of a symbol), a computation $x = exp$ or a delay d (in relative or physical time). The communication with the external environment are represented by $\sigma?$ with $\sigma \in \Sigma_{\text{in}}$ (reception of events) and $\sigma!$ with $\sigma \in \Sigma_{\text{out}}$ (emission of messages).

Moreover, all *branching* (multiple outgoing transitions from a single state) must have the form depicted in Figure 11[4]: A transition is fired from the state r as soon as

[4] IR can also contain conditional branchs that we do not describe here.

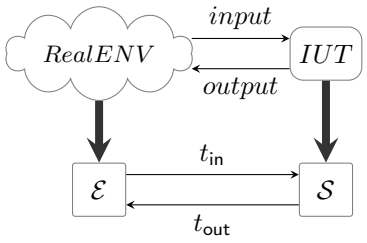

Figure 9: *Specification : reality (top) and models (down)*

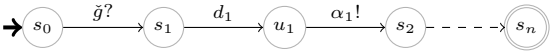

Figure 10: *IR of a loose and local group. In the initial state s_0, the automaton is waiting for the trigger symbol $\check{g}$. Once this symbol is read, it waits (in state s_1) for a delay d_1, and sends action α_1. Then it continues from state s_2 with the rest of the group.* □

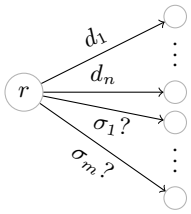

Figure 11: *Generic IR branching*

one of the delays d_i, $1 \leq i \leq n$ has expired (*i.e.* the time spent in r is d_i) or one symbol σ_j, $1 \leq j \leq m$ is received. We consider a synchronous model of time (following [8]): The time can flow only in source states of branchings. The other transitions, labeled by $\sigma!$ or $x = exp$ are instantaneous (i.e. a logical time 0 is spend in the source state).

The EFSMs composing a network are run concurrently: at each instant, every EFSM is in one control state. Initially, every EFSM starts in its *initial state*, which is unique.

Compiling Scores into IR (construction of the models)

The EFSM network $IR(s)$ is produced from a given score s by traversing the hierarchical structure of s. Intuitively, it contains one EFSM for each group in s and a fixed number of auxiliary EFSMs.

An EFSM called *error proxy* defines the notion of errors in the flow of musician events. To each $i \in \Sigma_{\mathsf{in}}$ we associate a unique new signal $\overline{i} \in \Sigma_{\mathsf{sig}}$, meaning that the event of id i (in the score s) is missing. The transitions of the error proxy are labeled by $i?$ (the event i has been detected) or $\overline{i}!$ (to signal that the event i is missing). Various definitions of the notion of missing events are allowed, and specified using parameters of the compilation command.

Next, we generate one EFSM for each group in the score. To a group g we associate 2 symbols denoted $\check{g}$ and $\bar{g}$. If g is a toplevel group, triggered by e_i, then $\check{g} = i \in \Sigma_{\mathsf{in}}$ and $\bar{g} = \overline{i} \in \Sigma_{\mathsf{sig}}$. Otherwise, $\check{g}$ and $\bar{g}$ are new signals of Σ_{sig}. The generic form of the EFSM of group g is depicted in Figure 12, where $init$ is the initial state and the sub-EFSMs $fsm(g)$, $mfsm(g)$ are defined according to the strategies for g (see examples in Figures 10 and 13).

Additionally, an EFSM $\mathcal{E}$ modeling the environment is constructed, in order to bound the space of possible interpretations of musicians considered for testing and avoid explosion during test input generation.

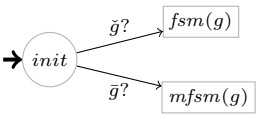

Figure 12: *EFSM associated to group g*

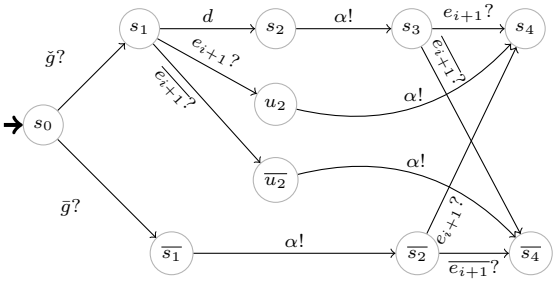

Figure 13: *IR of the beginning of a tight and global group with a unique action $\langle d, \alpha \rangle$. We have 4 execution modes, corresponding to the 4 lines: normal, earlier, missed-earlier and missed. In states s_0, s_3 and $\overline{s_2}$, the automaton is waiting for an event or group symbol. If this symbol is missed, it switches to the missed mode (resp. states $\overline{s_1}$, $\overline{s_4}$). In normal mode (state s_1), the automaton waits for a delay d before sending the action α. In missed mode (state $\overline{s_1}$), the automaton sends α without waiting. Moreover, in state s_1, the automaton waits concurrently for a delay d and for the detection of the next event e_{i+1} (at the current score position). If e_{i+1}, respectively $\overline{e_{i+1}}$, arrives before the expiration of d, then the automaton switches to mode earlier (state u_2), resp. missed-earlier (state $\overline{u_2}$). In both cases, the action α bound to the previous event e_i is sent without delay and then the automaton switches to normal or missed mode (s_4 or $\overline{s_4}$).* □

3.3 Timed Automata and Uppaal

Timed automata (TA) [14] are finite state automata extended with a finite set of real-valued variables called *clocks*. Every TA transition is labeled by a symbol (in a finite alphabet), and a linear constraint (*guard*) on the clock values: the transition can be fired only if the current values of the clocks satisfy the associated constraint. Moreover, every clock can be independently reset to 0 during a transition and keeps track of the elapsed time since the last reset. Some linear constraints (*guard*) on the clock values called *invariants* can also be attached to states, such constraint must be satisfied as long as the control stays in the state.

In a TA, all the clock values are expressed in a unique abstract time unit, the *model time unit* (mtu), *i.e.* all the clocks evolve at the same rate. The IR of a score s can be converted into an equivalent TA, under the restriction that: (i) all the delays are expressed in relative time in the score (like in all traditional scores) and (ii) the score contains no variables or all the variables can be evaluated statically.

Uppaal [5] is a symbolic model checker which permits users to write, simulate and verify timed automaton networks. The set of configurations of a TA $\mathcal{A}$ is infinite (it is the Cartesian product of the finite set of states of $\mathcal{A}$ and the infinite set of valuations of the clocks of $\mathcal{A}$). However, it is possible to transform a TA into a finite state automaton recognizing the same (untimed) sequences of symbols, using a finite equivalence on configurations (*region construction*) [14]. This fundamental technique gives a PSPACE algorithm for deciding reachability properties, implemented

[5] `http://www.uppaal.org`

efficiently in Uppaal.

Given an input trace t_{in} for testing and a TA model $\mathcal{S}$ of a score, it is possible to compute the corresponding output t_{out}, according to $\mathcal{S}$. A deterministic environment model $\mathcal{E}$ is first computed from t_{in} as in Figure 14. Then, a simulation is performed with Uppaal, on the automata network $\mathcal{E} \cup \mathcal{S}$, and t_{out} is obtained by tracing output during this simulation. The environment can be modified slightly in this process (from $\mathcal{E}$ presented in Figure 14), by introducing intervals on delays in transition's guards, in order to prevent state-space explosion in the generation of test cases (Section 3.4).

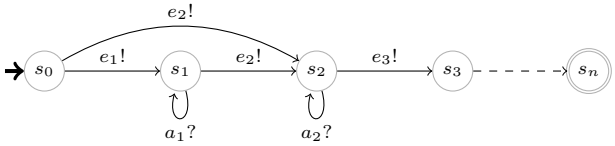

Figure 14: *An environment automaton*

3.4 Test Suite Generation with CoVer

Testing does not prove that Antescofo is crash-free, but the more test-case we have checked, the bigger guarantee we have. As we cannot test exhaustively for all possible performances on a given mixed score, a strategy is to consider a relevant set of test-cases (including extreme ones!) that covers in some sense the possible behavior of the IUT on the score. It is possible to generate automatically such sets of test-cases based on the formal specification of the system, and this problem has been extensively studied [15].

For this purpose, we use an Uppaal extension called CoVer [4], which has been used for testing Ericsson's industrial size networking systems [16]. It allows to generate test cases sets according to a user-written coverage criteria, defined as a finite state automaton called *observer* monitoring the execution of $\mathcal{S}$. The transitions of observers are labeled by Boolean predicates validated when some states or transitions in the TA model $\mathcal{S}$ have been reached. The model checker Uppaal is used to generate the input traces t_{in} enabling to reach a final state of a given observer for $\mathcal{S}$. This modular approach permits to target a specific group, or a specific problem such as error handling for testing, with a focus on Antescofo debugging.

Note that the traces t_{in} generated by CoVer do not contain tempo values, but only durations in relative time. They refer to a clock in the TA model which is not yet specified, and can be instantiated using an arbitrary time-map (for execution scenario of Section 2.4.1 or 2.4.3, first case), or by the detected tempo (Section 2.4.2 or 2.4.3, second case). Let us consider the generation of a test case on our running example (Figure 15). First, CoVer returns a couple of notes e_3 (E♮) and e_4 (D♯), with respective durations 0.27 (a little less than $\frac{2}{7}$) and 0.4 (a little less than $\frac{3}{7}$) beats. They represent time-shifts over the durations in the score. The other notes are assumed missing. Next, some arbitrary tempo values are generated: 60 bpm for e_3 and 114 bpm for e_4, completing an input trace t_{in}. The output trace t_{out} associated to t_{in} is then computed by Uppaal.

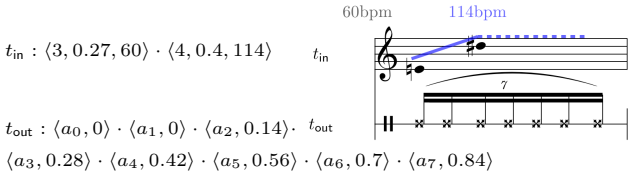

Figure 15: *Test input generation for the running example*

3.5 Execution and Verdicts

To summarize, based on an environment model $\mathcal{E}$ and a specification $\mathcal{S}$ compiled from an mixed score s, CoVer provides us with a covering suite of input traces $t^1_{\mathsf{in}}, \ldots, t^k_{\mathsf{in}}$ and the corresponding output traces $t^1_{\mathsf{out}}, \ldots, t^k_{\mathsf{out}}$. An execution with Antescofo on t^j_{in}, $1 \leq j \leq k$, following one of the scenarii of Section 2.4, will return real output traces t'^j_{out}. A step by step comparison of the t^j_{out} and t'^j_{out} will permit to draw a test verdict (see Section 2.5). A fixed error bound (approx. 0.1ms) is applied when comparing the delays, for dealing with latency.

The crucial point here is that when observers express that we cover all the edges of $\mathcal{E}$ and $\mathcal{S}$, then success on all the test cases generated by CoVer guarantees the conformance of the IUT to the model $\mathcal{S}$ of the score, *wrt* $\mathcal{E}$. This completeness result is obtained thanks to the use of the region construction in CoVer.

4. CONCLUSION

We have presented model based conformance testing approaches and their application to the SBIMS Antescofo. The generation of test input data and computation of the corresponding expected output is based on a formal model compiled from a mixed score, and done with the help of the symbolic model checker Uppaal.

The results obtained with these approaches, with real scores or small case studies, have permitted to identify and fix bugs in Antescofo. For instance, an erroneous cast of Antescofo's detected tempo caused the computation of wrong values of action delays. The small variation was detected when comparing t'_{out} against t_{out}.

The generation of input can be done automatically with CoVer, following covering criteria expressed as observers. This approach is oriented towards software engineering and debugging. Alternatively, the input can be produced manually by adding some fuzz to an ideal trace (using time-maps describing tempo variations and time shifts) as described at the end of Section 2.3. This gives no guaranty of coveredness but the input produced is musically more relevant (it corresponds to a performance). This approach is more oriented towards the preparation of concerts and can be helpful at composition time.

Another possible application of our framework is non-regression testing: An expected trace t_{out} can be simply recorded by monitoring an execution on a given t_{in} with a former reliable version of the SBIMS. Then one can compare with the trace t'_{out} produced on t_{in} by the new version under test. Of course, unlike model based approaches, this technique gives no guarantees that the execution t_{out} is

correct, it only permits to check automatically whether the new version behaves like the old one on t_{in}.

One limit of the approach is related to the input test data generated by CoVer, which tends to chose *shortest delays* for t_{in} inside regions. As a consequence, the tempo computed on this input can increase exponentially (since delays in t_{in} are expressed in relative time, refering to the current tempo). To avoid this problem, some other input delays (not the shortest) should be chosen in regions.
A second limitation is due to the expressiveness of TA. TA can have several clocks but they all run at the same frequency (the mtu). Hence, multirate is not supported in TA models whereas it is possible in Antescofo DSL. Moreover, in Antescofo DSL and IR, delays can be expressions with variables which cannot always be evaluated statically (e.g. when they depend on input). Some extensions of TA with variables are supported by Uppaal and remain to be studied in the context of IMS model-based testing.

The approach presented in Section 3 generates the test cases *offline*: the whole traces t_{in} and t_{out} are generated before test execution, which can be space consuming. Equivalent *online* approaches exist, with 'on the fly' generation and execution of traces. This is developed as Uppaal extension and named TRON. However it could not be used in our case due to a technical issue in the conversion from relative to physical time values. An interesting alternative could be to follow an approach similar to [17] combining fuzz testing and a white box approach. It consists in executing an online test loop, with on the fly random generation of test input from the code and, in parallel, the incremental developpement a propositional constraint (checked for satisfiability with a SAT solver), ensuring a form of coveredness.

Finally, a visualization of the output traces, with a graphical representation e.g. in Ascograph [6] would greatly benefit Antescofo's users, for composition assistance purposes. It would be useful e.g. to compare respective temporal positions of groups for different performances.

Acknowledgments

The authors wish to thank Antoine Rollet for his advises on MBT, and the team Ircam's Mutants for their help.

5. REFERENCES

[1] R. Rowe, "Interactive Music Systems: Machine Listening and Composing". AAAI Press, 1993.

[2] M. Krichen and S. Tripakis, "Black-box conformance testing for real-time systems", in SPIN'04, volume 2989, Springer, 2004, pp. 109–126.

[3] A. Hessel et al. "Testing Real-time systems using UPPAAL", in *Formal Methods and Testing (FMT)*, LNCS 4949, pp. 77–117, Springer, 2008.

[4] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson, "Specifying and generating test cases using observer automata", in *FATES'04*, 2004.

[6] http://forumnet.ircam.fr/product/antescofo/ascograph/

[5] A. Cont et al. "Correct Automatic Accompaniment Despite Machine Listening or Human Errors in Antescofo", in *ICMC 2012*.

[6] A. Cont, "A coupled duration-focused architecture for realtime music to score alignment", *IEEE Tr. on Pattern Ana. and Mach. Intel.*, 32(6), pp. 974–987, 2010.

[7] J. Echeveste, A. Cont, J.-L. Giavitto, and F. Jacquemard, "Operational semantics of a domain specific language for real time musician–computer interaction", *Discrete Event Dyn. Sys.*, 23(4), pp. 343–383, 2011.

[8] J. Echeveste, J.-L. Giavitto, and A. Cont, "A Dynamic Timed-Language for Computer-Human Musical Interaction," INRIA, RR-8422, 2013.

[9] T. A. Henzinger, "Two challenges in embedded systems design: predictability and robustness", *Phil. Tr. of The Royal Society* 366(1881), pp. 3727–3736, 2008.

[10] R. B. Dannenberg, "Abstract time warping of compound events and signals," *Computer Music Journal*, 21(3), pp. 61–70, 1997.

[11] H. Honing, "Structure and interpretation of rhythm and timing," *Dutch Journal of Music Theory*, 7(3), pp. 227–232, 2002.

[12] T. A. Henzinger, C. M. Kirsch, "The Embedded Machine: Predictable, Portable Real-time Code", *ACM Trans. Program. Lang. Syst.* 29(6), 2007.

[13] F. Jacquemard, C. Poncelet. "Antescofo Intermediate Representation". INRIA research report, 2014.

[14] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Computer Sci.*, vol. 126, pp. 183–235, 1994.

[15] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, "Model-Based Testing of Reactive Systems". Springer Verlag, 2005.

[16] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using uppaal," in *Formal methods and testing*, Heidelberg: Springer-Verlag, 2008, pp. 77–117.

[17] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production". MSR-TR-2012-55, 2012.